Sample Method1™ Testing Metrics

Problem Rate - This metric measures the number of problems over time.

The actual problem rate should be tracked against the planned problem rate through testing to confirm assumptions, identify problems, and verify that the cumulative incoming rate is sufficiently flat prior to release. The problem rate serves as a prediction tool for scheduling test and fix activities.

Calculation:

Number of problems/ Number of days of execution

Definitions:

Problem: Any nonconformity discovered during the stage. Includes errors associated with the current development stage and defects associated with a previous development stage. This also includes problems with the test model.

Target:

Below or at the planned problem rate. A numerical target has not been recommended here, because the problem rate will vary by project, team, architecture, tool, and complexity. The value of the target should fluctuate, depending on the amount of test execution that is planned for the week and which pass is being executed (fewer problems should be identified in the later passes).

Analysis Steps and Corrective Actions:

- Analyze the rate at which problems are being discovered.
- Measure rework defects. If the number of defects is greater than estimated, it may be because one of every two defects fixed is breaking something else, resulting in more defects. Monitoring the repair effectiveness metric will answer this question. See Metric 7. Repair Effectiveness Percentage for targets and courses of action.
- Implement well-defined processes. A well-defined process will reduce the number of defects. A well-defined process reduces the number of defects built into the product, provides mechanisms for discovering them early if they do occur, and incorporates a mechanism to facilitate continuous improvement to reduce the number of defects. Keep in mind that everyone involved must be educated on the process and fully understand it for the process to be effective.
- Implement standards. Implementing standards for specification and test deliverables in addition to programming reduces the number of defects created, because it allows the developer more time to concentrate on the requirements, design or test content rather than on the format.
- Implement formal inspections. Studies have shown that formal inspections are an efficient way to discover problems early, preventing them from being discovered later as defects. Inspections concentrate on standards, process, content, and traceability, and are conducted for review of specifications, code and test plans. The purpose of inspections is to find as many (legitimate) problems as possible in order to contain problems to the development stage that created them. Once a deliverable is inspected and the problems discovered in the inspection are corrected, the deliverable provides a better starting point for the next development stage, thereby reducing the chance of problems and defects appearing in subsequent development stages.

Reporting:

Calculate and review the number of problems found every week to ensure that problems are identified and corrective action taken in a timely manner. During product test execution, this metric may need to be calculated daily/hourly to determine if product test is still on schedule.

This metric should be tracked against the planned problem rate. The actual problem rate curve should flatten out close to the end of product test, indicating that the product is ready for release (see the example below). The actual problem rate should continue to be compared to the planned problem rate through operational readiness test and benefits realization test. However, the number of problems should not increase dramatically during these tests. Finally, the problem rate should be calculated once the solution has been in production for 1 to 2 years, indicating the true quality of the solution.

Inspection Return on Investment (ROI) Ratio - How effective is the inspection process at discovering problems?

This metric illustrates the savings and costs associated with inspections. Inspections cause a significant amount of time and money to be spent early in the process. This investment can continue to be justified only by understanding the savings it provides.

The purpose of inspections is to find problems early in the process, because they are more costly to fix if they are discovered as defects later in the process. This metric relates the savings to the cost in order to illustrate the true monetary savings provided through inspections.

The inspection ROI ratio is influenced by the effectiveness of inspections (errors found per hour of inspection) and the average time to fix errors and defects.

Calculation: Number of errors found * (average hours to fix a defect – average hours to fix a problem) at a given stage / average hours of inspection at a given stage

Definitions:

Error: Most problems discovered during an inspection are errors, assuming that the problems relate to the work completed in the current development stage. A missing requirement detected in a design application architecture inspection is not an error, but a defect.
Average hours to fix a defect: Total hours spent fixing and retesting defects for a given stage / Number of defects fixed for a given stage.
Average hours to fix an error: Total hours spent fixing and retesting errors for a given stage / Number of errors fixed for a given stage.

Hours of inspection for that stage: Time spent by all participants in every inspection for that development stage. Includes the time of the scribe and moderator as well as all other participants. In addition, include:

- Time spent by the developer to prepare for the inspection, including time spent on copying, distribution, and other logistics
- Time spent by the participants to read the material before the inspection
- Time spent documenting problems and meeting minutes from the inspection

Do not include:

- Time spent developing the specification or test plan to be inspected
- Time spent fixing problems, which arose from the inspection

Target:

External research indicates that a good return on investment is greater than 2.0, but less than 10.0. A return on investment greater than 10.0 would indicate that the original build process or the process to fix defects is inadequate. If the value of this metric is greater than the target range, it should only be used for management purposes. It should not be communicated to the inspection participants, as they should concentrate on finding problems and not worry about finding too many.

Reporting:

By definition, return on investment should be calculated weekly throughout the development stages using the previous release's average hours or the estimated average hours to fix a defect.

The report templates located in the Testing Metrics Starter Kit show the inspection return on investment on both a week-to-week and a cumulative basis. The week-to-week report shows fluctuations in productivity and the effect improvements to the process have on the metric. The cumulative data shows overall project rates.

It is better to show the cumulative reports to inexperienced report viewers, as the peaks and valleys in the week-to-week report may cause concern to the inexperienced report viewer. The cumulative value is smoother, but still shows the effects of the week-to-week fluctuations; these fluctuations will not be of the same magnitude. The downfall to showing the rates cumulatively is that earlier rates may affect all rates and falsely skew the curve.

Zage Design Metric - To further Refine/Improve the Process and Product Quality by determining which modules are error prone (at the end of the design application architecture stage)

This metric measures the complexity of the module at the end of the design application architecture stage.

During the testing process (especially in the later stages of testing), the most problematic modules are often those that were the most difficult to design and build or were not carefully designed. A complex design may be difficult for the programmer to fully understand; therefore, these modules are often the last ones to be finished. As a result, the modules become critical path, and there is rarely time to go back to redesign and build; when these modules undergo testing, they tend to result in numerous defects. This scenario does not stop with testing. Often the system is put into production, and the same modules that had problems during testing now have problems in production, making them an absolute maintenance nightmare.

A proactive approach to this problem is to use the Zage Design Metric. Although this metric must be derived manually (unless a structured - formal - specification language is used), it can easily be derived early enough in the process to facilitate redesign, schedule more time to build and test, or assign the most experienced resources to build and test these modules. This metric is derived at the end of the design application architecture stage for a module. By counting the number of interfaces to other modules and the amount of data that is handled by the given module, a complexity count is derived. The premise is that the more interfaces a module has and the more data it handles, the more complex it is to build, design, and test, and therefore the more error prone it becomes. This complexity count is viewed in the context of the counts for the other modules, so attention is focused on those modules that are furthest above the average.

Identify Product Quality Problems Early - Research has shown that it is preferable to identify quality problems early to allow management ample time to respond in order to meet time and budget commitments. This metric can be used to predict software quality before costly implementation has occurred.
The difficulty of testing and maintaining complex modules should help the customer understand that complex modules need to be redesigned in order to ensure they are maintainable.

Calculation:

For each module, calculate:

(Fan in * Fan out) + (Data in * Data out)

Once this is calculated for each module, find the mean by summing all the module figures and dividing by the number of modules. The most complex modules will be located two standard deviations from the mean; the higher the value, the more complex the module. It is recommended that the top 25% of the modules as determined by this complexity metric also be reviewed, even though they do not fall out of the two standard deviations. This will focus attention on modules that fall outside the mean that may also be complex and difficult to test.

Definitions:

Fan in: The number of modules that call this module.

Fan out: The number of modules called by this module.

Data in: The number of data elements or data structures coming into this module. A data structure is a group of related data elements. For example, an address is a data structure composed of data elements for the street, city, state, and zip code. This metric cannot be calculated using data elements for some modules and data structures for others, because this will make the metric inaccurate. Consistently use either data structures or data elements throughout.

Data out: The number of data elements or data structures flowing out of this module.

Module: The smallest piece of executable application code that makes up an entire solution.

Target:

All modules fall within 1 standard deviation from the mean as determined by external research.

Reporting:

Calculate the actual problems by module weekly to ensure that problems are identified and corrective action taken in a timely manner.

Testing Metrics: McCabe's Cyclometric Complexity

Further Refine/Improve the Process and Product Quality by determining which modules are error prone (at the end of the design automated processes and the generate and code work units stages).

McCabe's Cyclomatic Complexity - This metric provides a complexity measure based on the number of paths through a module.

Ideally, the Zage Design Metric was calculated for each module during the design application architecture stage and all complex modules were redesigned. However, this is usually not enough, because subsequent decisions and changes may have changed the complexity of the modules. Module complexity should be reconfirmed after the design automated processes stage and again after the generate and code work units stage. This can be accomplished through McCabe's Cyclomatic Complexity metric, which is based on the premise that the more paths through a program, the more complex it is. Although this metric is manually derived during the design automated processes stage, it can easily be derived early enough in the process to facilitate redesign. This metric is derived through a tool during the generate and code work units stage, and it should be compared to the count taken at the end of the design automated processes stage.

The benefit of this metric is that complex/error-prone modules are identified early enough to enable appropriate action to be taken: redesign of complex modules, allocation of more time to build and test complex modules, identification of problem areas to be focused on during inspections, or assignment of more experienced personnel to build and test complex modules.
Determine Component Test Cycles Required - Another benefit of this metric is that because it is based on the number of paths through a module, it can be used to determine how many test cycles are required for component test, thereby providing 100% path coverage. This can serve as an entry criteria to assembly test.

Provide an Objective Determination of Complexity - Finally, this metric is objective. A design/module reviewed twice by the same person or by more than one person will always have the same value, thus eliminating the chance of differing interpretations.

Calculation:

Count the number of "if" statements and the number of "or" operands in the module. The equation would become n + 1, where n is the number of "if" statements and "or" operands. The one is added because there would still be one path even if there were no "ifs." For example:

```
Example #1      IF  (A > B OR B > C)
                     D = YES
                ELSE
                     D = NO
Example # 2     IF  A  THEN B
                          ELSE    IF C THEN B
                          ELSE    D
```

The following is an interpretation of cyclomatic complexity:

Cyclomatic Complexity	Interpretation
less than 5	simple and easy to understand
less than or equal to 10	not too difficult
greater than 20	highly complex
greater than 50	untestable

(Source: Applied Software Measurement by Capers Jones)

Analysis Steps and Corrective Actions:

- Analyze complex modules starting first with the most complex modules.

There may be complex modules that require no action. For example, a program that alphabetizes data will have 26 decisions, but will actually be very simple to code and test. The value of this metric triggers investigations into the modules with the highest cyclomatic complexity or cyclomatic complexity greater than 20, and forces a conscious decision to be made as to whether or not take corrective action.

Review the modules with the highest cyclomatic complexity or cyclomatic complexity higher than 20 and take the following actions if necessary:

- Redesign all complex modules. This will ensure that these modules are easier to build, test, and maintain.

- Allow more time for building and testing complex modules. Studies have been conducted which indicate that complex modules will be more difficult to build and test. Therefore, more problems should be anticipated in these modules, and more time should be planned to build and test these modules. The higher the complexity, the more time should be allowed for building and testing. Keep in mind that the best alternative is to redesign these modules, but if redesign is not an option, allowing more time to build and test will mitigate the risk.

- Assign more experienced personnel to build and test complex modules. Assigning more experienced personnel to build and test these modules will minimize the risk. Keep in mind that the best alternative is to redesign these modules, but if redesign is not an option, assigning more experienced personnel will mitigate the risk.

Reporting:

Report this metric on a weekly basis to determine which modules are complex and should be redesigned early enough in the process to have a positive cost impact.